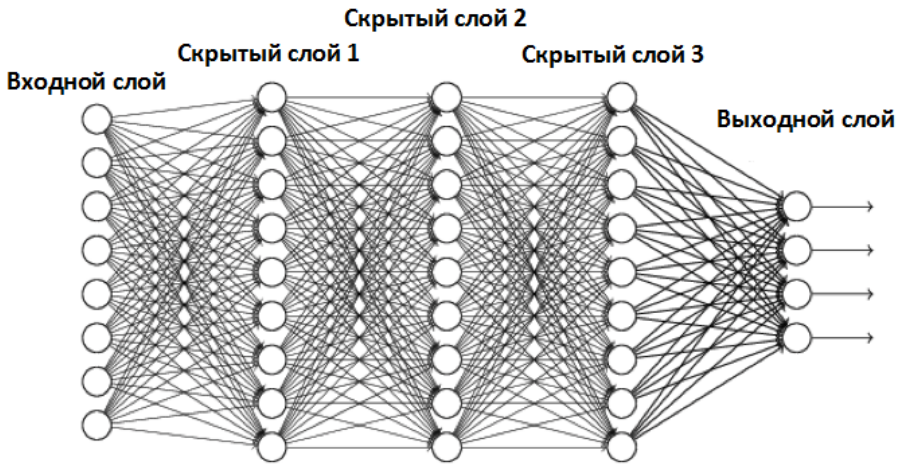


Настройка параметров SVM

Дата публикации Sep 6, 2019

Машина опорных векторов является одним из самых популярных алгоритмов классификации. Подход SVM к классификации данных элегантен, интуитивен и включает в себя очень классную математику. В этом руководстве мы подробно рассмотрим различные параметры SVM, чтобы понять, как мы можем настраивать наши модели.

Прежде чем мы сможем развить наше понимание того, что делают параметры, мы должны понять, как работает сам алгоритм.

Как работают SVM

Машины опорных векторов работают, находя точки данных разных классов и рисуя границы между ними. Выбранные точки данных называются опорными векторами, а границы называются гиперплоскостями.

Алгоритм рассматривает каждую пару точек данных до тех пор, пока не найдет ближайшую пару из разных классов, и не проведет прямую линию (или плоскость) на полпути между ними.

Если входные данные линейно разделимы, то решение для гиперплоскости является простым. Но часто бывает так, что области классификации пересекаются, и ни одна прямая плоскость не может выступать в качестве границы.

Один из способов обойти это - проецировать ваши данные в более высокие измерения, создавая дополнительные функции. Вместо двумерного пространства вы получаете от наличия особенностейа также*б,*Вы могли бы объединить их (например,ab, a^2, b^2)и попытаться найти закономерности (или разделительную гиперплоскость) в этих измерениях.

Но есть проблема с этим подходом. Хотя наличие дополнительных измерений облегчает поиск гиперплоскости, оно также дает вашему алгоритму больше возможностей для изучения.

SVM позволяют нам обойти это дополнительное обучение, используя трюк с ядром.

Трюк с ядром

Возможно, вы слышали о ядре трюк. Это одна из тех вещей, которая включена в каждое обсуждение SVM, но часто это не объясняется так, как могло бы быть.

Ядро - это просто функция, которая принимает две точки данных в качестве входных данных и возвращает оценку сходства. Это сходство можно интерпретировать как показатель близости. Чем ближе точки данных, тем выше сходство.

Крутая вещь в функциях ядра заключается в том, что они могут дать нам оценки сходства из более высоких измерений без необходимости трансформации наших данных.

Мы можем найти самые близкие точки данных в гораздо более высоких измерениях без их фактического присутствия. Это означает, что мы можем получить все преимущества от дополнительных функций, не разрабатывая и не изучая их.

Таким образом, фокус ядра заключается в использовании функции ядра вместо дорогостоящего преобразования.

(Если вы математически настроены, вам может быть интересно, какие функции можно использовать в качестве ядра. Ответ заключается в том, что неявная функция, которая дает нам оценку сходства в более высоких измерениях, существует всякий раз, когдапространству внутреннего произведения может быть дана мера, которая является положительно определенной.)

Ядра SVM

Теперь, когда мы понимаем, как работают SVM и какова хитрость ядра, мы можем перейти к ноутбуку Jupyter и посмотреть, как наш выбор ядра влияет на наши модели.

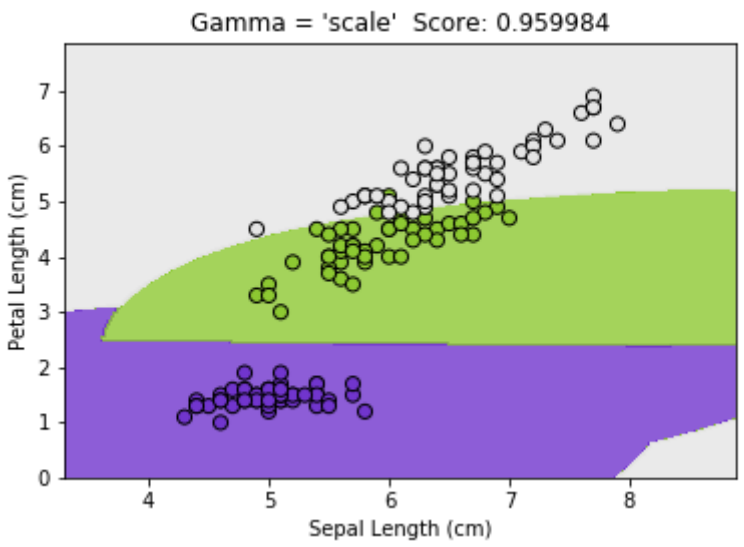
Если вы используете Scikit-Learn, вы увидитедокументациячто вы можете выбрать из нескольких разных ядер при создании объекта классификатора векторов поддержки (SVC). Они включают:

Давайте загрузимНабор данных радужной оболочки.

```
from sklearn import datasets iris = datasets.load_iris()
```

Для наглядности оставим только первые две функции набора данных (длина Sepal и ширина Sepal). Мы можем построить их и использовать цвета, чтобы показать их класс (вид).

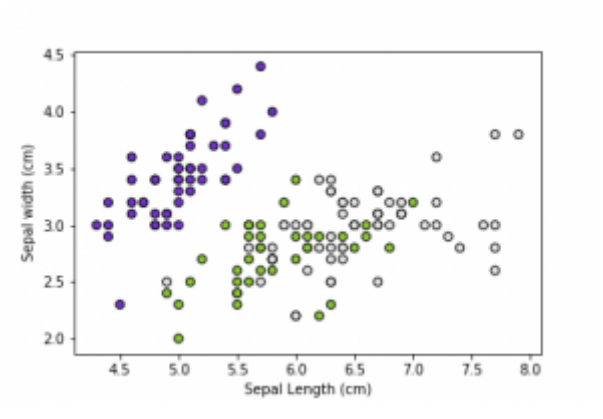
```
sepal_length = iris.data[:,0]
sepal_width = iris.data[:,1]
plt.scatter(sepal_length, sepal_width, c=iris.target)
```



Теперь у нас есть визуальное понимание наших данных, и мы можем перейти к тому, как ядра SVM влияют на границы классов.

Линейное ядро

```
from sklearn.svm import SVC
clf = SVC(kernel='linear')
clf.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf,
sepal_length, sepal_width)
```

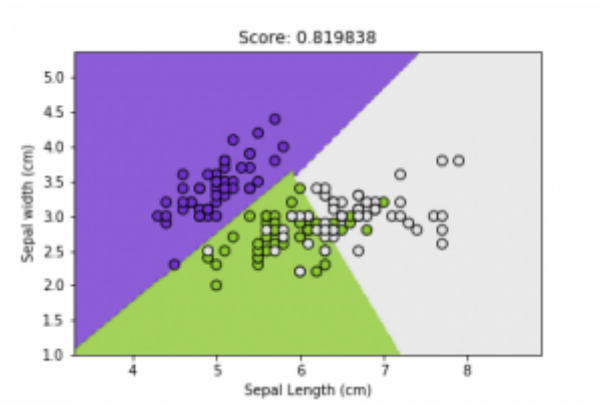


Линейные ядра вычисляют сходство во входном пространстве. Они неявно определяют трансформацию в более высокие измерения. Из-за этого каждая из гиперплоскостей на рисунке выше представляет собой прямые линии.

Возможно, вы видели графики SVM, на которых выглядит, как будто границы решения изогнуты (как те, которые появляются). Важно отметить, что границы решений изогнуты только в области ввода. В неявном пространстве пространственных объектов они представляют собой прямые линии или плоскости. (На самом деле, подобные графики не создаются с помощью уравнений для извилистых линий, они создаются путем классификации сетки крошечных точек и раскраски их в соответствии с выводом SVM, как вы можете видеть из функции create_grid_plot.)

Ядро RBF

```
clf = SVC(kernel='rbf')
clf.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf,
sepal_length, sepal_width)
```

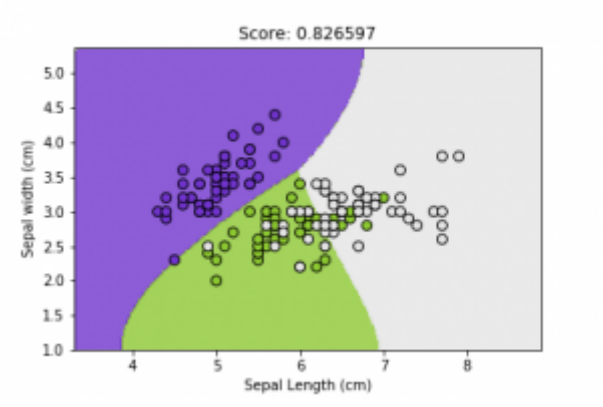


[Радиальная базисная функция](#)Ядро не просто помогает нам избежать некоторых дополнительных функций. Пространство объектов RBF имеет бесконечное количество измерений. Это означает, что мы можем использовать ядро для построения очень сложных границ принятия решений. Чем больше измерений, тем больше шансов, что мы найдем гиперплоскость, которая аккуратно разделяет наши данные.

*радиальная основа*Часть названия происходит от того факта, что эта функция уменьшается в цене, когда она удаляется от центра. (Центр в данном случае является опорным вектором.) Это объясняет, почему границы решений имеют форму колокольчика, когда мы их визуализируем

Полиномиальное ядро

```
clf = SVC(kernel='poly')
clf.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf,
sepal_length, sepal_width)
```



На самом деле мы уже встретили ядро полинома. Линейные ядра являются частным случаем полиномиальных ядер, где степень = 1.

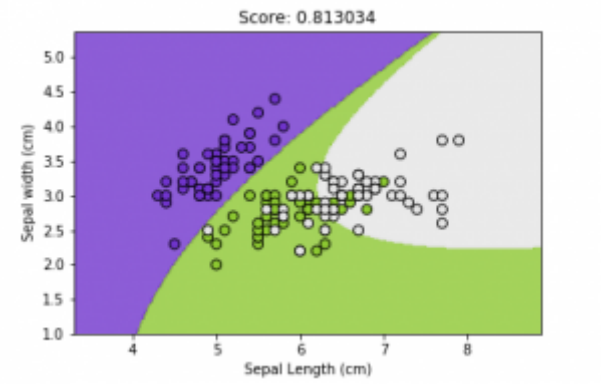
Ядро полинома позволяет нам изучать шаблоны в наших данных, как если бы у нас был доступ к функциям взаимодействия, которые являются результатом объединения ранее существующих функций (a^2, b^2, ab и т. Д.)

Если вам интересно, как выбирать между полиномом и RBF: RBF - ваш лучший выбор, если вы не работаете в НЛП, где квадратичные (степень = 2) полиномы имеют тенденцию работать хорошо.

Сигмоидальное ядро и попарные метрики

До сих пор каждое ядро работало достаточно хорошо, без дополнительной настройки параметров или изменения функций, но это все изменится. Давайте посмотрим на сигмовидное ядро.

```
clf = SVC(kernel='sigmoid')
clf.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf,
sepal_length, sepal_width)
```



Использование сигмовидного ядра позволило нашей модели предсказать один и тот же класс для каждой входной строки, что привело к показателю точности около 17%. Так что же случилось?

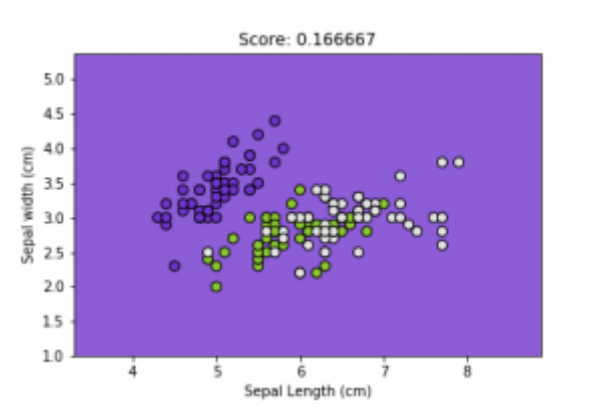
Мы говорили о том, что функции ядра - это всего лишь меры сходства. Если у вас возникли проблемы с моделью SVM, может быть полезно запустить ваши данные через ядро, чтобы увидеть, произойдет ли что-нибудь непредвиденное. Удобно, что каждое из ядер, доступных для классификатора векторов поддержки, включено в Scikit-Learn.попарные метрикимодуль. Это означает, что мы можем передать наши данные только ядру и посмотреть, что мы получим. Давайте попробуем это для сигмовидного ядра.

```
from sklearn.metrics.pairwise import sigmoid_kernel sigmoid_kernel(np.c_[sepal_length,
sepal_width])
>>>
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

Этот вывод выглядит подозрительно для меня. Ядро сигмоиды говорит, что каждая точка одинаково похожа на любую другую точку - неудивительно, что SVM испытывает трудности с отрисовкой границ классов!

Если вы изучали глубокое обучение (или логистическую регрессию), вы узнаетесигмовидная функция, Служит активацией - вкл или выкл, ноль или единица. Вы также можете знать, что входные данные, которые не находятся между нулем и единицей, могут вызвать хаос. Давайте попробуем измерить наши особенности диафрагмы и посмотрим, какое влияние это окажет.

```
from sklearn.preprocessing import normalize
sepal_length_norm = normalize(sepal_length.reshape(1, -1))[0] sepal_width_norm =
normalize(sepal_width.reshape(1, -1))[0] clf.fit(np.c_[sepal_length_norm,
sepal_width_norm], iris.target) create_grid_plot(clf, sepal_length_norm,
sepal_width_norm)
```



Успех. Наш показатель точности вернулся к более чем 75%.

C: параметр штрафа

Что делает параметр C в классификации SVM? Он сообщает алгоритму, насколько вы заботитесь о неправильно классифицированных баллах.

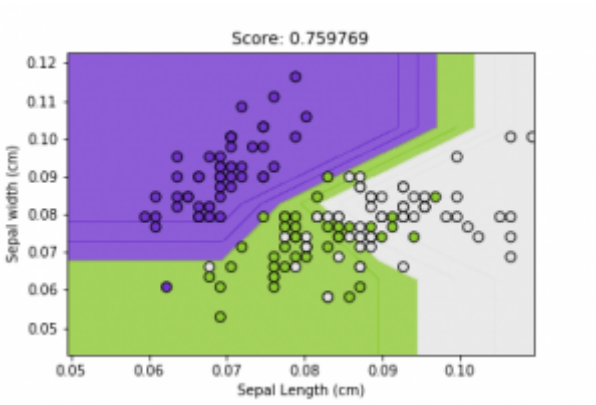
Как правило, SVM стремятся найти гиперплоскость с максимальным запасом. То есть линия, которая имеет как можно больше места с обеих сторон.

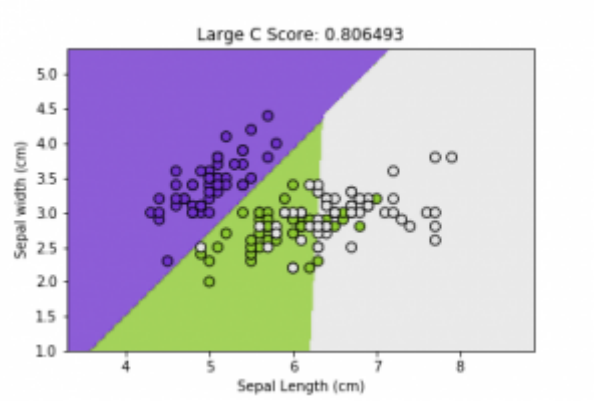
Высокое значение C говорит алгоритму, что вам важнее правильно классифицировать все тренировочные точки, чем оставлять пространство для маневра для будущих данных.

Подумайте об этом так: если вы увеличиваете параметр C, вы держите пари, что данные тренировок содержат максимально экстремальные наблюдения. Вы держите пари, что будущие наблюдения будут дальше от границ, чем точки, на которых вы обучали модель.

Давайте посмотрим, как изменение параметра C может повлиять на наши модели.

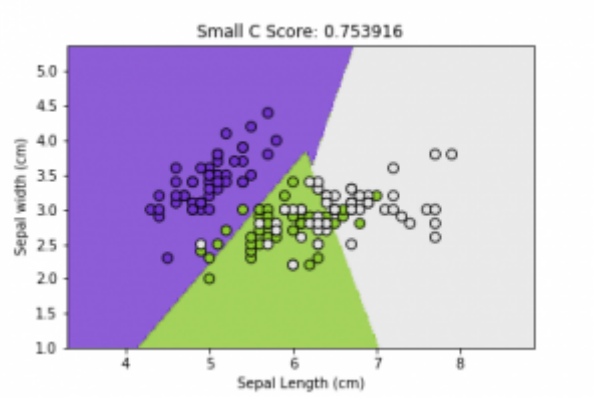
```
clf1 = SVC(kernel='linear', C=1000000)
clf1.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf1,
sepal_length, sepal_width)
clf2 = SVC(kernel='linear', C=.0000001)
clf2.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf2,
sepal_length, sepal_width)
```

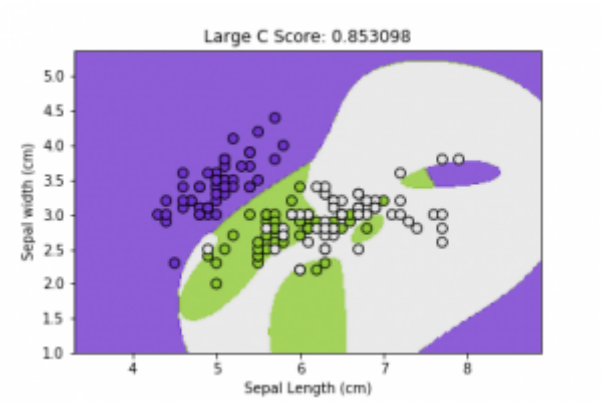




Вы можете видеть, что большее значение C создало более четкие границы между областями классификации. Давайте посмотрим, как это влияет на ядро RBF.

```
clf1 = SVC(kernel='rbf', C=1000000)
clf1.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf1,
sepal_length, sepal_width)
clf2 = SVC(kernel='rbf', C=.0000001)
clf2.fit(np.c_[sepal_length, sepal_width], iris.target) create_grid_plot(clf2,
sepal_length, sepal_width)
```

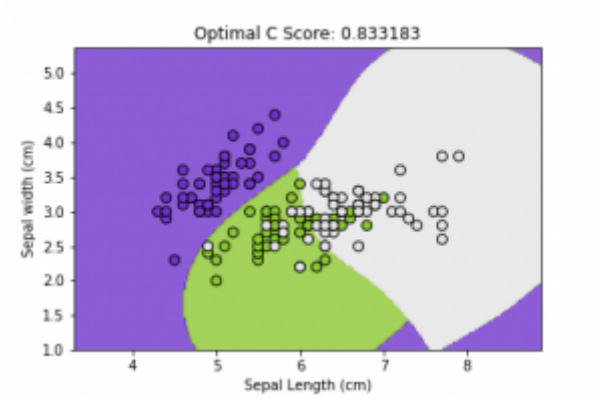


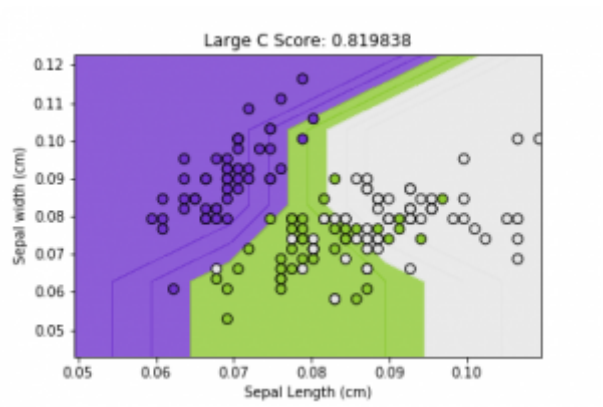


В сочетании с ядром RBF (или гауссовским) большие значения параметра C могут значительно превысить данные. Это вызвало отдельные области классификации на первом рисунке выше. Если мы бежимGridSearchCVпо параметру C мы находим, что идеальное значение для C равно 10:

Переоснащение и изолированные области исчезли, в результате чего мы получили показатель точности 83%, что на 1% больше, чем в нашей исходной модели ядра RBF.

Прежде чем перейти к другому параметру, вот как большие и малые значения параметра C влияют на ядро сигмоида:

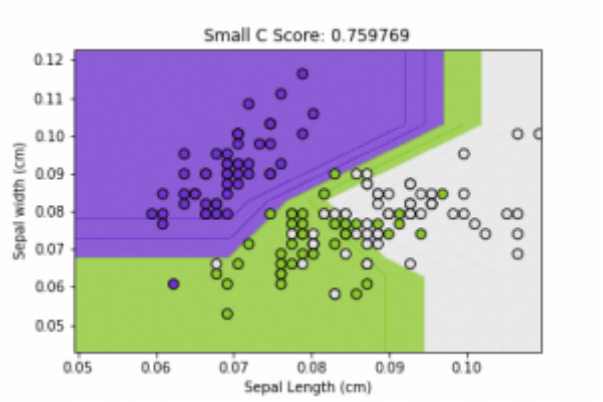




Вы можете видеть, что мы повысили показатель точности нашей (изначально) очень плохой сигмовидной модели примерно до той же точности, что и другие ядра, просто увеличив параметр C. И именно поэтому мы настраиваем параметры!

Гамма в СВМ

Чтобы все перемешать, давайте изменим функции, с которыми мы работаем. Вместо длины чашелистика и ширины чашелистика, давайте использовать длину чашелистика и длину лепестка и взглянем на наши данные:

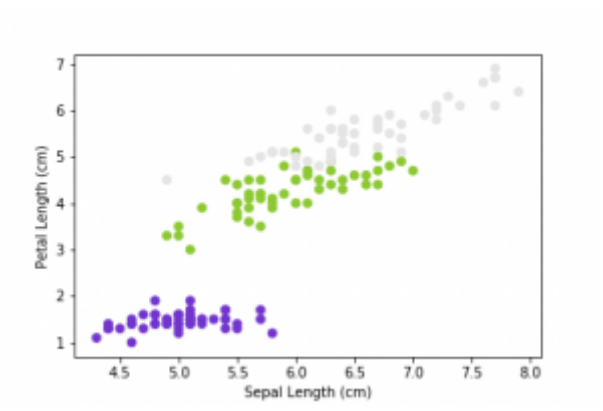


(Мы видим, что разные классы кажутся более отделимыми, чем при использовании предыдущих функций, но зеленый и серый виды по-прежнему пересекаются.)

Гамма-параметр имеет наиболее интуитивный смысл, когда мы думаем о ядре RBF (или гауссиане). Как я упоминал выше, границы классов Гаусса рассеиваются по мере удаления от опорных векторов. Гамма-параметр определяет, как быстро происходит это рассеяние; большие значения уменьшают эффект любого отдельного вектора поддержки.

Приведенный выше сюжет делает увеличение гаммы отличной идеей. В конце концов, если бы мы могли просто уменьшить влияние каждого из этих кластеров, мы могли бы получить четко очерченные границы:

```
clf = SVC(kernel='rbf', C=10, gamma=100)
clf.fit(np.c_[sepal_length, petal_length], iris.target) create_grid_plot(clf,
sepal_length, petal_length)
```

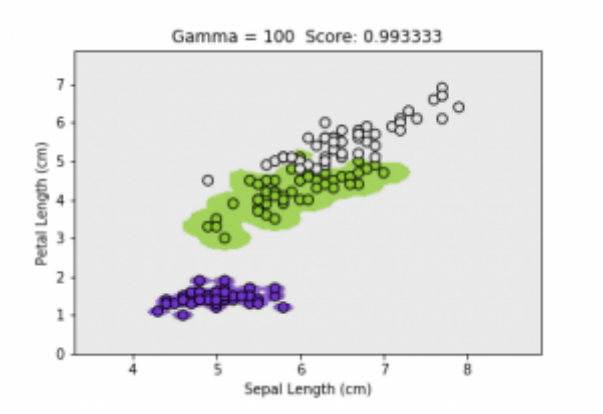


Точность 99,3% - это не плохо, правда?

Однако важно помнить, что мы еще не разбили данные на обучающие и тестовые наборы и не сделали никаких предположений относительно будущих данных. Если мы выберем высокое гамма-значение, подобное этому, для нашей окончательной модели, мы должны быть уверены, что все будущие экземпляры попадут в крошечные области, которые мы определили.

В качестве проверки здравомыслия, давайте сделаем 500 раундовслучайная выборкаи оценим стабильность нашей модели:

```
scores = []
for i in range(0,500):
    X_train, X_test, y_train, y_test = train_test_split(np.c_[sepal_length,
petal_length], iris.target)
clf = SVC(kernel='rbf', C=10, gamma=100)
clf.fit(X_train, y_train) scores.append(accuracy_score(clf.predict(X_test), y_test))
plt.hist(scores)
```

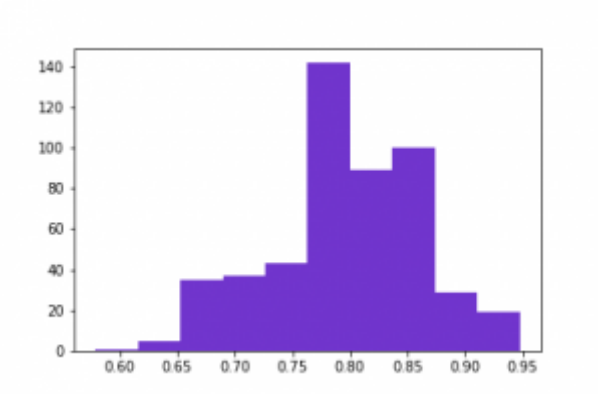


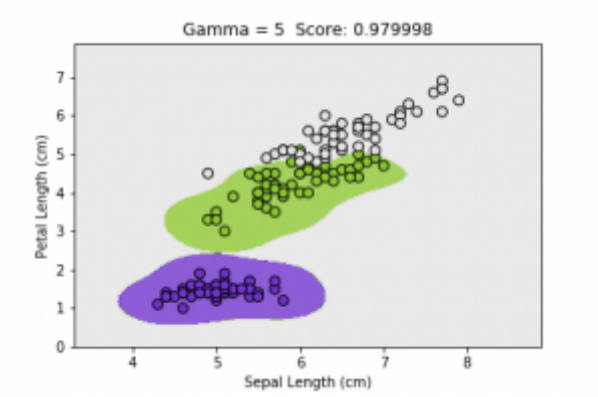
Глядя на эту гистограмму, мы видим, что не только точность 99,3%, которую мы достигли при использовании всех данных, никогда не встречается на небольших выборках, большинство оценок попадают в диапазон, аналогичный моделям без гамма-настройки, и многие из них значительно

улучшаются. хуже.

Рафинирующая Гамма

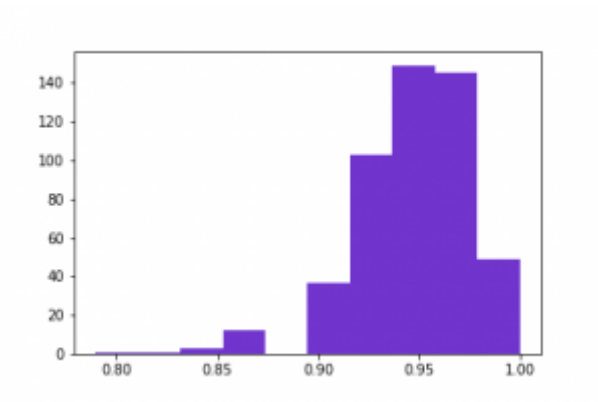
Давайте попробуем более разумное значение для гаммы. Вот что мы получаем, когда устанавливаем параметр равным 5:

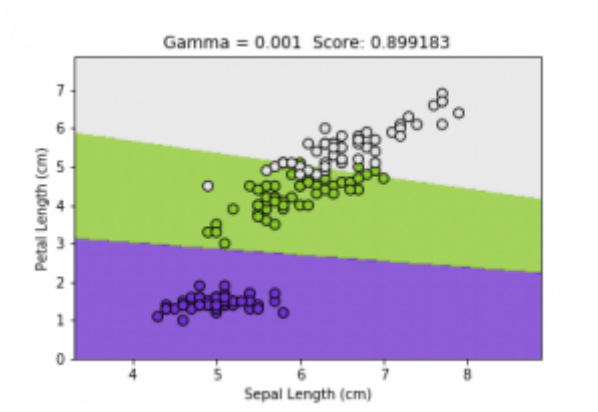




Влияние опорных векторов еще больше возрастает с сильно уменьшенной гаммой, и взамен мы получаем результаты, которые в среднем намного лучше, чем у ненастроенных моделей. Более того, мы на самом деле получили несколько тестов в тесте точности 97,5–100%.

Давайте снова серьезно уменьшим гамма-параметр, на этот раз до 0,001, и посмотрим, что происходит с границами наших решений.

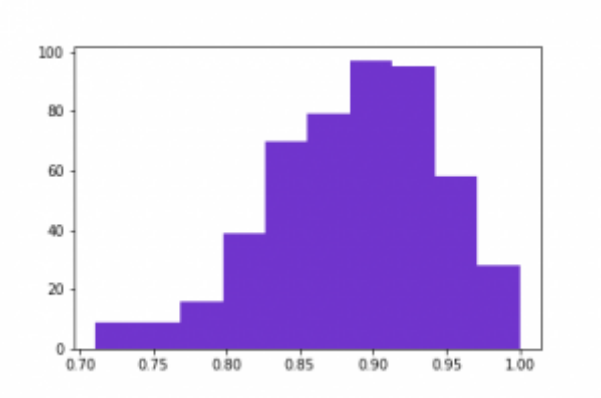


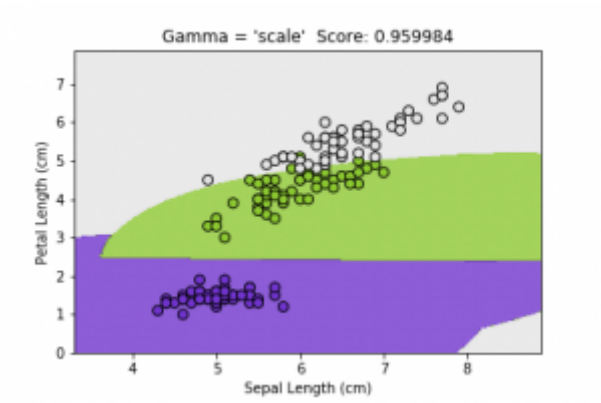


Оценка модели значительно упала, и области пространства признаков более четко классифицированы. С другой стороны, 500 итераций показывают, что производительность модели в среднем улучшается по сравнению с ненастроенными моделями (которые колебались около 80%).

Текущее значение по умолчанию для гаммы Scikit-Learn - «auto». Это принимает значение 1 / n_features, в нашем случае это будет 1/2 или 0,5. В более поздних версиях sklearn значением по умолчанию будет «scale», которое принимает значение 1 / (n_features * X.var ()), которое в нашем случае будет около 0,168. Давайте попробуем «масштабный» подход:

```
1 / (2 * np.c_[sepal_length, petal_length].var())
>>> 0.16804089263919647 clf = SVC(kernel='rbf', C=10, gamma=(1 / (2 *
np.c_[sepal_length, petal_length].var())))
clf.fit(np.c_[sepal_length, petal_length], iris.target) create_grid_plot(clf,
sepal_length, petal_length)
```





Это дает нам точность 96% и не ограничивает области классификации. Гистограмма показывает, что подвыборки также дают хорошую точность. Таким образом, настройка «масштаба», кажется, является отличным выбором для набора данных Iris.

Резюме

Как и при любой настройке параметров, вы должны смотреть не только на те изменения, которые приносят вам высочайшую точность, а на те, которые наиболее полно отражают проблему, над которой вы работаете.

Вы должны оценить, используя свой собственный опыт:

- Будущие наблюдения могут быть сгруппированы вместе
- Классы должны иметь большие поля между ними
- Кажущиеся кривые в границах решения происходят из-за случайного шанса или ошибки измерения

Я надеюсь, что вы нашли это исследование параметров SVM полезным. Мы рассмотрели различные типы ядра, попарные функции, параметр штрафов и назначение гаммы. Все это должно помочь вам уточнить и настроить ваши будущие модели машин опорных векторов.

Если у вас есть какие-либо вопросы, пожалуйста, оставьте комментарий.

Спасибо за чтение!

Оригинальная статья

- Фреймворки и библиотеки (большая подборка ссылок для разных языков программирования)
- Список бесплатных книг по машинному обучению, доступных для скачивания
- Список блогов и информационных бюллетеней по науке о данных и машинному обучению
- Список (в основном) бесплатных курсов машинного обучения, доступных в Интернете